Derived Features for EMF by Integrating Advanced Model Queries*

István Ráth, Ábel Hegedüs, Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
{rath,hegedusa,varro}@mit.bme.hu

Abstract. When designing complex domain-specific languages, metamodels are frequently enriched with *derived features* that correspond to attribute values or references (edges) representing computed information in the model. In the popular Eclipse Modeling Framework, these are typically implemented as imperative Java code.
In the paper, we propose to integrate the EMF-IncQuery model query framework to the Ecore metamodeling infrastructure in order to facilitate the efficient and automated (re-)computation of derived attributes and references over EMF models. Such an integration allows to define derived features using an expressive graph-based model query language [1], and offers high performance and scalability thanks to the incremental evaluation technique of EMF-IncQuery [2]. In addition, our approach offers to automate two typical associated challenges of EMF tools: (1) values of derived features are immediately recalculated upon model changes and (2) notifications are sent automatically to other EMF model elements to report changes in derived features.

1 Introduction

The design of complex domain-specific languages (e.g. in the automotive or avionics domains) frequently necessitate the use of advanced metamodeling techniques. Metamodels are complemented with *well-formedness constraints*, which enable the validation of the consistency of instance models with respect to such constraints, thus allowing to spot design flaws early in the development process. *Derived features*, which correspond to attribute values or references (edges) that represent computed information in the model, also proved to be useful in complex metamodeling scenarios. For instance, they frequently serve as auxiliary (helper) functions when implementing model simulators, and they also allow to compact the storage of the model.

In the popular Eclipse Modeling Framework (EMF), these derived features are most often implemented as user-defined algoritms computed by imperative

* This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) project, the grant TÁMOP (4.2.2.B-10/1–2010-0009) and the János Bolyai Scholarship.

Java code. Unfortunately, (1) most existing techniques re-calculate values of derived features in EMF models on-demand (i.e. when corresponding getters are called), which hinders integration into user interfaces where changes in the values of derived features should immediately be reflected. Furthermore, (2) it is challenging to properly implement notification propagation between (a chain of) derived features upon value changes, which is necessary when components or model elements are required to depend upon a derived feature. Finally, (3) as the calculation of derived features is always started from scratch (not taking previous computations and changes into account), it is also challenging to implement complex queries in Java in a way that does not severely impact the overall performance.

The advanced model query framework EMF-IncQuery has proved to be efficient in the incremental re-validation of well-formedness constraints over large models [2] scaling up to millions of elements[1]. Its expressive, declarative graph-based query language offers high level of reuse in queries [1]. In the paper, we propose to seamlessly integrate the EMF-IncQuery framework to the Ecore metamodeling infrastructure, in order to facilitate the efficient and automated computation of derived attributes and references over EMF models.

Our proposed approach, which is fully implemented and documented[2], offers to automate the entire workflow of developing derived features in EMF. In the approach, (1) derived features are defined using an expressive graph-based model query language and are calculated by an algorithm that (2) listens to all *incoming notifications* that impact on the computation, (2) issues *outgoing notifications* when the value of the derived feature changes, (3) keeps an *up-to-date cache* that is refreshed based on incoming notifications and used for computing outgoing notification. Finally, (4) since outgoing notifications may cause incoming notifications, the algorithm also *stabilizes such notification loops*.

In the rest of the paper, Section 2 provides a brief overview on derived features in EMF models. Then, we propose to use a graph based model query language to define derived features for EMF in Section 3. Section 4 provides a detailed architecture and core algorithms to synthesize notifications for derived features based upon incremental query evaluation. Additional issues for seamless integration to the EMF infrastructure are discussed in Section 5. Finally, Section 6 overviews related work and Section 7 concludes our paper.

2 Derived Features in EMF

Derived features in EMF models represent information that can be calculated from other model elements and typically represent an aggregate view of the model. Essentially, we distinguish between *derived attributes* and *derived refer-*

[1] The current paper does not include performance specific contributions to the EMF-IncQuery framework, we kindly refer the reader to `http://viatra.inf.mit.bme.hu/performance` for additional details.

[2] `http://viatra.inf.mit.bme.hu/incquery/examples/derivedfeatures`

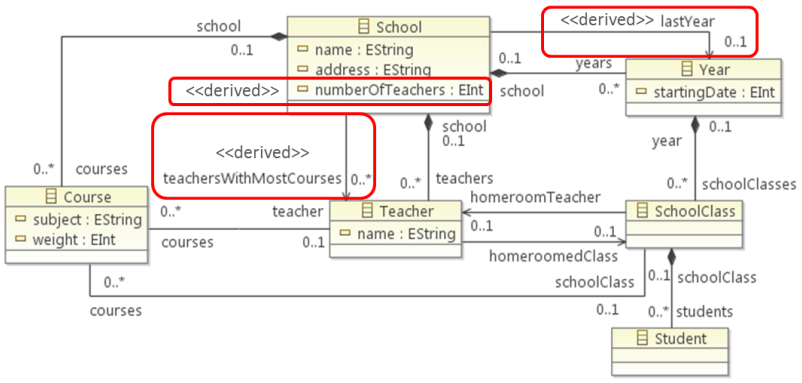


Fig. 1. The metamodel of the Schools domain

ences (representing "virtual" connections between model elements). In our example, both are represented graphically by the derived stereotype in Figure 1.

In the current paper, we illustrate our approach on a simple demonstration domain of Schools (encoded in EMF's Ecore language as illustrated in Figure 1) that manage Courses involving Teachers, and enroll their students assigned to Years and SchoolClasses. The metamodel contains simple EAttributes (like e.g. name of a Teacher or the startingDate of the school Year) and regular EReferences such as the school of a Teacher. More importantly for the sake of this paper, it also contains three *derived features*:

- numberOfTeachers is a derived attribute of School representing a counter for the total number of Teachers belonging to the School as represented by the corresponding school EReference;
- lastYear is a derived reference from School to Year, and points to the last academic Year stored in the model, which can be calculated from the startingDate of all Years;
- teachersWithMostCourses represent the busiest teachers of the School, i.e. those who teach the most courses.

Derived features in EMF are not maintained explicitly in instance models, but calculated on-demand by hand-written code. These calculations are frequently supported by ad-hoc Java implementations integrated directly into the EMF model representation, which significantly reduces the portability and compatibility of the metamodel.

Unfortunately, developers may encounter additional key challenges when aiming to use derived features in EMF models:

- **Performance.** Depending on the complexity of the semantics of derived features, their evaluation may impose a *severe performance impact* (since

complex calculations and extensive model traversal may be necessary for execution). Note that this scalability issue is especially important when derived feature values need to be re-evaluated many times and will affect all other software layers using the model code, including the user interface, model transformations, well-formedness validators etc.
- **Notifications.** Due to the *difficulty of propagating notifications* for derived features, derived features are typically re-evaluated on demand. This may also manifest as model changes not (properly) triggering user interface updates. Note that EMF defines the notifications for derived features as well, however, it is the programmer's responsibility to create notifications. Since the values of derived features are usually not cached, proper notifications including the old values (e.g. setting a single value or removing from a list) are hard to implement. Furthermore, notifications of one derived feature may cause new notifications, leading to notification loops, the programmer must ensure that these are stabilized in order to avoid infinite loops.

Our proposal, namely, the integration of an advanced model query framework EMF-IncQuery provides a solution for all of these challenges using a high-level graph-based query language for defining derived value calculations. As the performance characteristics of the EMF-IncQuery engine have been shown to be agnostic of query complexity and model size [2], derived features of complex semantics and inter-dependencies can be used without severe evaluation performance degradation. Additionally the update propagation mechanism of EMF-IncQuery (using *delta monitors* [3]) will be connected to the EMF Notification layer so that the application software components are automatically kept up-to-date about the value changes of derived features.

3 Definition of Derived Features as Model Queries

We now propose to use the graph pattern based model query language of EMF-IncQuery as the specification language for derived features of EMF models. Therefore a brief introduction to this query language is provided first, followed by a detailed description on how this general purpose query language is adapted to specify derived features.

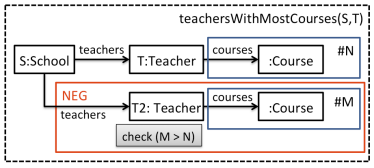
3.1 Model queries by graph patterns: an overview

Graph patterns [4] are an expressive formalism used for various purposes in model-driven development, such as defining declarative model transformation rules, capturing general-purpose model queries including model validation constraints, or defining the behavioral semantics of dynamic domain-specific languages. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of *structural constraints* prescribing the existence of nodes and edges of a given type, as well as *expressions* to define *attribute constraints*. A *negative application*

condition (NAC) defines cases when the original pattern is *not* valid (even if all other constraints are met), in the form of a negative sub-pattern. A match of a graph pattern is a group of model elements that have the exact same configuration as the pattern, satisfying all the constraints (except for NACs, which must not be satisfied). The complete query language of the EMF-IncQuery framework is described in [1], while several examples will be given below.

3.2 Derived features as model queries

Sample derived features First, we demonstrate on an example how the graph pattern teachersWithMostCourses(S,T) (Figure 2) can be used to express the calculation of the derived EReference teachersWithMostCourses (connecting *School* and *Teacher* in Figure 1), that is, to identify those teachers who have the maximum number of *Course* instances assigned (through the *Teachers.courses* reference).




```
1  pattern teacherWithMostCourses(S, T)=
2  {
3   School.teachers(S,T);
4   neg pattern moreCourses(S,T) = {
5    Teacher.courses(T,C) # N;
6    School.teachers(S,T2);
7    Teacher.courses(T2,C2) # M;
8    check(M > N);
9   }
10 }
```

Fig. 2. Model query to define teachersWithMostCourses in graphical and textual syntax

This model query formulated as a graph pattern has two parameters: S and T, denoting the source and the target end of the derived EReference. The query defines the designated set of teachers by combining a NAC and cardinality constraints. It expresses that a teacher T belongs to this set if and only if there is no other teacher $T2$ whose number of courses M (calculated by counting the number of elements connected along the *courses* reference) would be larger than the number of courses N (counted as before) of teacher T. The right side of Figure 2 shows the corresponding textual syntax.

Model queries for derived features numberOfTeachers and lastYear are defined similarly in Figure 3. The definition of the latter contains some additional interesting language elements.

- The modifier shareable prescribes that different (but type consistent) pattern variables are allowed to be bound to the same model elements (e.g. $D1$ and $D2$ can be bound to the same date element).
- Y =/= Y2 checks that the two model elements bound to variables Y and $Y2$ are different.
- Using the find keyword, graph patterns are allowed to reuse other graph patterns. Therefore, if a derived feature is defined as a model query by a corresponding graph pattern, this derived feature can be reused in other queries,

```
pattern numberOfTeachers(S,N)=
{
 School.teachers(S,T) # N;
}
```

```
pattern lastYear(S,Y)= {
 find years(S,Y);
 neg shareable pattern laterYear(S,Y)= {
   find years(S,Y);
   find startingDateOfYear(Y,D1);
   find years(S,Y2);
   find startingDateOfYear(Y2,D2);
   check(D1 < D2);
   Y =/= Y2;
} }
```

Fig. 3. Model queries for numberOfTeachers and lastYear

and thus, in other derived features. In fact, we will discuss in Section 5 that even legacy derived features (defined by Java code) can participate in such usage with appropriate notification mechanisms.

Derived features can be defined as model queries using the graph pattern based language of EMF-IncQuery if the following three well-formedness rules are met by corresponding query definitions:

1. *Each graph pattern should have exactly two parameters.* In case of derived attributes, the first parameter denotes the corresponding EClass of the attribute, while the second parameter denotes the value of the parameter itself (see numberOfTeachers). In case of derived references, the first parameter denotes the source (i.e. the container EClass) while the second parameter denotes the target of the EReference.
2. *First parameter: always input.* General model queries allow the same pattern to be used with either input or output parameters (i.e. parameter bindings can be carried out at execution time), in case of derived features, the first parameter (referring to the container) should always be an input parameter, which is a bound to a type-compliant contextual EMF object (e.g. S is bound in all three graph patterns above). This restriction is conceptually equivalent to the context element of an OCL constraint.
3. *Restrictions on result set.* In case of a derived features with explicit lower and upper bounds (e.g. 1..* or 0..1), the result set of the model query should comply with these restrictions. While upper bounds can be enforced by omitting results, the violation of lower bound is logged only as warnings.

In the actual query language, rules 1 and 2 are can be satisfied either by using exactly two query parameters, or by using *pattern annotations* for multi-parameter queries that explicitly specify which of the parameters is the context and which one will correspond to the target (or value). Furthermore, the adherence to all three rules are checked at editing time by a built-in query language validator in the EMF-IncQuery tooling. In summary, the modular nature of the EMF-IncQuery language aims to allow the language engineer to construct a library of cross-referencing queries without copy-paste reuse.

4 From Incremental Query Evaluation to Notifications for Derived Features

In this section, we outline how the incremental query features of the EMF-IncQuery framework are integrated to notification-based applications in transparent way, by mapping changes of the results sets to notification objects for derived features. We present an architectural overview and an algorithm to carry out this mapping.

4.1 Incremental evaluation of queries

The key to efficient evaluation and change notification for derived features is the incremental graph pattern matching infrastructure of the EMF-IncQuery framework (introduced in [3]). The internal architecture is shown in Figure 4.

The input for the incremental graph pattern matching process is the EMF instance model and its notification API. Callback functions can be registered through this API for instance model elements that receive notification objects (e.g. ADD, REMOVE, SET etc.) when an elementary manipulation operation is carried out.

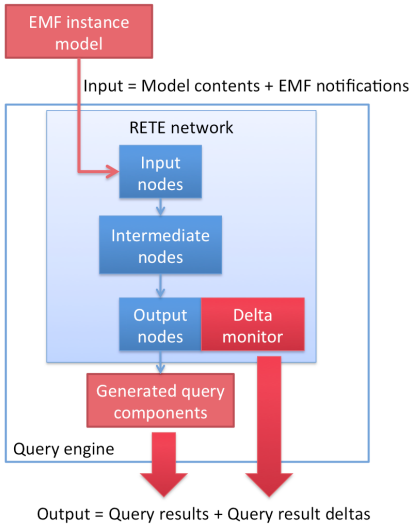


Fig. 4. The EMF-IncQuery architecture

Based on a query specification, EMF-IncQuery constructs a Rete rule evaluation network [3] that processes the contents of the instance model to produce the query result at its output node. Query results are then post-processed by *auto-generated query components* to provide a type-safe access layer for easy integration into applications. This Rete network remains in operation as long as the query is needed: it continues to receive elementary change notifications and propagates them to produce *query result deltas* through its *delta monitor* facility, which are used to incrementally update the query result. These deltas can also be processed externally, which is a key feature for the integration of derived features (Section 4.2).

By this approach, the query results (i.e. the match sets of graph patterns) are continuously maintained as an in-memory cache, and can be retrieved directly. Even though this imposes a slight performance overhead on model manipulation, and a memory cost proportional to the cache size (approx. the size of match sets), EMF-IncQuery can evaluate very complex queries over large instance

models very efficiently. These special performance characteristics [2] address the scalability challenge (Section 2) as long as enough memory is available, as they allow EMF-IncQuery-based derived features to be evaluated incrementally, even for complex queries over large instance models.

4.2 Integration architecture

To support derived features, the outputs of the EMF-IncQuery engine are to be integrated into the EMF model access layer at two points: (1) *query results* are provided in the getter functions of derived features, and (2) *query result deltas* are processed to generate EMF Notification objects that are passed through the standard EMF API so that application code can process them transparently. The overall architecture of our approach is shown in Figure 5.

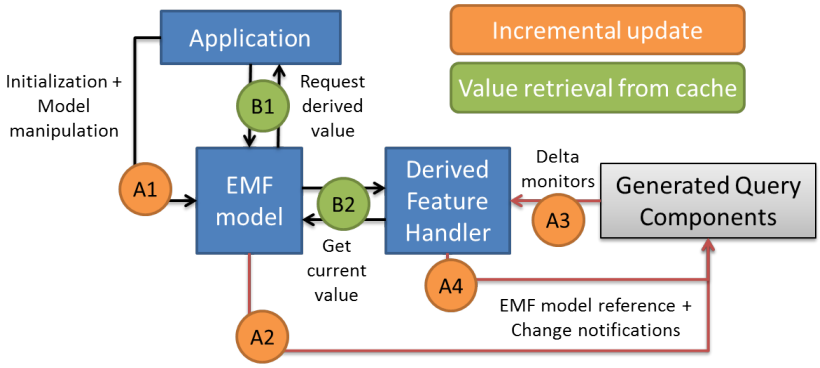


Fig. 5. Overview of the integration architecture

The application accesses both the model and the query results through the standard EMF model access layer – hence, no modification of application source code is necessary. In the background, as a novel component type, *derived feature handlers* are attached to the EMF model plugin that integrate the generated query components (pattern matchers). This approach follows the official EMF guidelines of implementing derived features and is identical to how ad-hoc Java code, or OCL expression evaluators are integrated.

When an EMF application intends to read a derived feature (B1), the current value is provided by the corresponding derived feature handler (B2) by simply retrieving the value from the cache of the related query. When the application modifies the EMF model (A1), this change is propagated to the generated query components of EMF-IncQuery along notifications (A2), which may update the delta monitors of the derived features (A3). Changes of derived features may in turn trigger further changes in the results sets of other derived features (A4).

Illustrative example Figure 6 illustrates a detailed elaboration EMF-IncQuery feature handlers, which process elementary model manipulation notifications to update, and generate notifications for derived features. The figure corresponds to a case where the user created a new Teacher for a School through the Editor which is essentially a School.getTeachers().add(teacher) method call on the Model. During the add method, the School EObject sends an ADD notification to the Notification Manager, which will notify the EMF-IncQuery Query Engine about the model modification. The Query Engine updates the match sets of each query and registers the match events in the Deltamonitor. Once it's finished with updating the Rete network, it invokes the callback method of each IncqueryFeatureHandler. Each handler has a Deltamonitor from which it retrieves the found and lost match events since the last callback to processes them. During the processing, the handler may send notifications of its own that are propagated to listeners. Anytime the derived feature value is retrieved from the model (e.g. getNumberOfTeachers), the handler is accessed for the current value of the feature, which is returned directly.

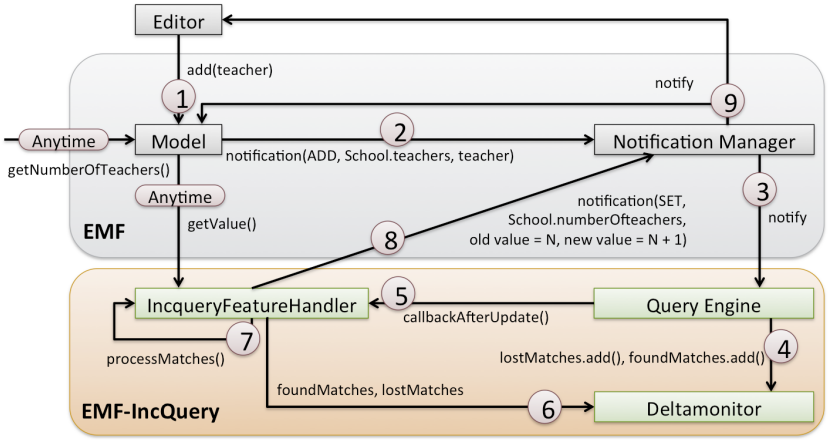


Fig. 6. Elaboration of the execution

4.3 From changes of match sets to notifications

We now explain the notification processing and propagation procedure in algorithmic detail. For the sake of simplicity, we introduce an auxiliary discriminator variable *Kind* whose value represents three distinct cases:

- SINGLE and MANY correspond to derived references of target multiplicity 1 and *, respectively (lastYear and teachersWithMostCourses in Figure 3);
- COUNTER corresponds to the simplified case where a value of the derived attribute is defined as the match set size of a query (see numberOfTeachers in Figure 3).

– More complex derived feature kinds with an arbitrary, deterministic iteration algorithm can also be handled by the approach.

The main part of our derived feature handler algorithm is an event loop that is called by the EMF-IncQuery query engine each time the underlying Rete network is updated as a result of some model manipulation (see Algorithm 1).

Algorithm 1 Main event loop

```
1:  let S ← Source, F ← Feature, DM ← DeltaMonitor, k ← Kind                ▷ Input variables
2:  let (k = SINGLE)?iV ← null : (k = COUNTER)?iV ← 0 : iV ← ∅             ▷ Internal value init
3:  let pU ← null, N ← ∅                                                   ▷ Global variables
4:  function EVENTLOOP
5:      let pU ← null
6:      let found ← processFoundMatches(DM.matchFoundEvents)               ▷ Processing found events
7:      let DM.matchFoundEvents ← DM.matchFoundEvents \ found              ▷ Removing events
8:      let lost ← processLostMatches(DM.matchLostEvents)                  ▷ Processing lost events
9:      let DM.matchLostevents ← DM.matchLostevents \ lost                 ▷ Removing events
10:     if partialUpdate ≠ null then      ▷ Stored value not yet used, handle partial match event
11:         let N ← N ∩ notification(SET, null, pU)
12:         let iV ← pU                                                    ▷ Updating value
13:     end if
14:     while N ≠ ∅ do                                                    ▷ Notification sending loop
15:         let n ← N[0]
16:         let N ← N \ n
17:         S.eNotify(n)                                  ▷ Sending notification through source
18:     end while
19: end function
```

The algorithm is initialized with the following input variables (line 1): (1) the EObject *Source* whose derived feature is handled; (2) the derived *Feature*; (3) the *DeltaMonitor* for the query matcher; and (4) the previously mentioned discriminator value *Kind*. Each handler stores an internal value for the feature, initialized in line 2 depending on *Kind*. Finally, the handler uses two global variables: *pU* for storing partial events and the set N of unsent notifications.

The event loop starts from line 4, it first resets the partial event store, then processes matches found since the last execution of the loop (line 6). These events are supplied by the delta monitor of the query and removed after processing is finished. Similarly, the matches lost since the last execution are also processed (line 8) and removed after. When a derived feature with SINGLE kind is used and only a match-found event occurs without a match-lost event, an additional processing step is required to handle the partial event (line 11). This occurs when the query did not lose any matches since the last event loop, but a new match is found. This translates to a notification representing the setting of the feature value from *null* to *pU* (line 12). Finally, if there are any unsent notifications (line 14), the first notification n in the list N is sent through the *Source* EObject. By separating the notification sending from the calculation of the derived feature value, the notification loop is stabilized, since new notifications caused by n are simply added to the list N, which will be depleted after all, if causal circularity between the definitions of derived features is avoided.

New matches The handling of match-found events is detailed in Algorithm 2. The processFoundMatches function iterates through the match-found events (line 3), and extracts the target object from the event (line 5), if the source EObject of the event equals *Source*. Depending on the *Kind* of the feature, a notification is created and the internal value is updated (line 7 for COUNTER and line 12 for MANY). For SINGLE kind features, the target object is stored for later usage (line 10). Finally, the list of processed events is returned.

Algorithm 2 Processing match-found events

```
1:  function procesFoundMatches(events)
2:      let P ← ∅
3:      for all e ∈ events do
4:          if e.source = S then
5:              let target ← e.target                    ▷ Extracting feature target from event
6:              if k = COUNTER then
7:                  let N ← N ∩ notification(SET, iV, iV + 1)
8:                  let iV ← iV + 1                      ▷ Updating value of repeating algorithm
9:              else if k = SINGLE then
10:                 let pU ← target                      ▷ Storing value for later processing
11:             else if k = MANY then
12:                 let N ← N ∩ notification(ADD, null, target)
13:                 let iV ← iV ∩ target                 ▷ Updating value
14:             end if
15:         end if
16:         let P ← P ∩ e
17:     end for
18:     return P
19: end function
```

Lost matches The handling of match-lost events is similar to the processing of match-found events, see Algorithm 3. The processLostMatches function iterates through the match-lost events (line 3), and extracts the target object from the event (line 5), if the source EObject of the event equals *Source*. Depending on the *Kind* of the feature, a notification is created and the internal value is updated (line 7 for COUNTER and line 14 for MANY). For SINGLE kind features, the stored value of *pU* is used for creating the notification (line 10). Finally, the list of processed events is returned at the end of the function.

Summary In summary, the combined pattern matching and notification processing process ensures that EMF-IncQuery-based derived features behave exactly as normal features of EMF instance models. This addresses the final, integration-related challenge of Section 2), by ensuring that user interfaces, model validators etc. can safely depend on such derived features, without on-demand querying.

5 Integration Issues with EMF Tooling

5.1 Integration with Ecore

In the prototype implementation of our proposal, we integrated our approach to the EMF Tooling by a code generator that supports the automatic generation

Algorithm 3 Processing match-lost events

```
1: function processLostMatches(events)
2:     let P ← ∅
3:     for all e ∈ events do
4:         if e.source = S then
5:             let target ← e.target                    ▷ Extracting feature target from event
6:             if k = counter then
7:                 let N ← N ∩ notification(SET, iV, iV − 1)
8:                 let iV ← iV − 1                      ▷ Updating value of repeating algorithm
9:             else if k = single then
10:                let N ← N ∩ notification(SET, target, pU)        ▷ Using stored value
11:                let iV ← target                                  ▷ Updating value
12:                let pU ← null                                    ▷ Resetting stored value
13:            else if k = many then
14:                let N ← N ∩ notification(REMOVE, target, null)
15:                let iV ← iV \ target                             ▷ Updating value
16:            end if
17:        end if
18:        let P ← P ∩ e
19:    end for
20:    return P
21: end function
```

of integration code for our components (EMF-IncQuery *derived feature handlers*). The input of the code generator is a simple generator model (referencing the EMF genmodel for the domain) that crosslinks derived features with EMF-IncQuery query specifications (which are stored as EMF models thanks to the Xtext2-based tooling).

```
    teachersWithMostCoursesHandler = IncqueryFeatureHelper.createHandler(
            this,
            SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
            TeacherWithMostCoursesMatcher.FACTORY,
            "School",
            "Teacher",
            FeatureKind.MANY_REFERENCE);
 * @generated NOT
 */
public EList<Teacher> getTeachersWithMostCourses() {

    if(teachersWithMostCoursesHandler != null) {
        Collection<Object> temp = teachersWithMostCoursesHandler.getManyReferenceValue();
        return new UnmodifiableEList<Teacher>(this,
                SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
                temp.size(), temp.toArray());
    } else {
        return new UnmodifiableEList<Teacher>(this,
                SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
                0, null);
    }
}
```

Fig. 7. Sample generated code for derived feature handler instantiation and getter

The generated integration code (Figure 7) consists of (a) the instantiation of derived feature handlers (in the constructor of EObjects), which ensures that

their lifecycle is tied to the hosts, to enable their garbage collection together with the instance model itself; (b) getter implementations that delegate calls to the appropriate function of the feature handler object, and wrap the result in unmodifiable ELists to ensure that any attempt to write to derived features will result in a runtime exception.

5.2 Integration with legacy Java code for derived features

In practice, a complete refactoring of an EMF-based tool to exclusively use EMF-IncQuery-based derived features might not be realistic. Hence, we implemented an additional *derived feature adapter* (Figure 8) as a lightweight add-on component for EMF model plugins, which can be used to augment existing derived feature implementations (regardless of whether Java or OCL is used).

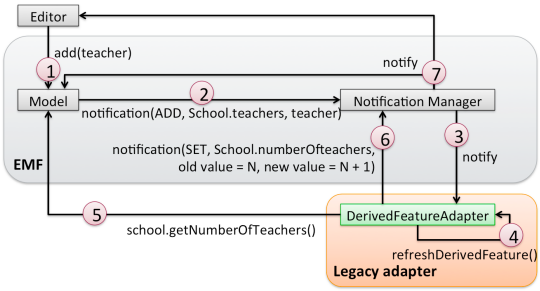


Fig. 8. Derived feature handlers

The basic concept motivated by a suggestion in the Eclipse FAQ[3] is analogous to the previous discussion. The language engineer can add a few lines of Java code to the generated EMF model plugin: these derived feature adapters attach listeners (through the EMF Notification API) to the (explicitly specified) features a derived feature depends on, and receive notifications when model changes are registered (steps 1-2-3 in Figure 8). These notification objects are then processed and converted into new notification objects for the derived feature, propagating through the manager to application code (steps 4-5-6-7 in Figure 8).

This approach has additional key advantages: (1) notification support can be added – with a small implementation effort – to "legacy" derived features, without having to re-write them in EMF-IncQuery; (2) queries specified in EMF-IncQuery (whether for derived features, or on-the-fly validation purposes, or within model transformations) can reference derived features seamlessly.

[3] `http://wiki.eclipse.org/EMF/Recipes#Recipe:_Derived_Attribute_Notifier`

6 Related Work

Model queries over EMF. There are several technologies for providing declarative model queries over EMF, e.g. EMF Model Query 2 [5] and EMF Search [6]. Other graph pattern based techniques like [7,8] have been successfully applied in an EMF context. But none of these support incremental evaluation, therefore they cannot be used for integrating derived features in the way we proposed.

OCL evaluation approaches. OCL [9] is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of OCL, the project Eclipse OCL provides a powerful query interface that evaluates OCL expressions over EMF models. However, backwards navigation along references in EMF can still have low performance [2], which may influence the performance of OCL evaluation without additional support.

Aiming at incremental evaluation, the impact analysis (IA) approach for OCL constraints [10] is functionally similar to our approach (but conceptually different in terms of underlying incremental algorithm) in using change notifications to identify constraints that should be re-evaluated, although it does not cache partial matches. An added feature of our approach is to automatically provide notifications for derived features (which could be – but currently is not – implemented for OCL tools). As future work, we aim to compare IA and our approach and even combine the benefits of our current implementation with the benefits of existing OCL-based solutions.

Cabot et al. [11] present an approach for incremental runtime validation of OCL constraints and uses promising optimizations, however, it works only on boolean constraints, and as such it is less expressive than our technique.

An interesting model validator over UML models [12] incrementally re-evaluates constraint instances whenever they are affected by changes, however the approach is only applicable in environments where read-only access to the model can be easily recorded, unlike EMF. Additionally, the approach is tailored for model validation, general-purpose model querying is not viable.

Balsters [13] presents an approach for defining database views in UML models as derived classes using OCL. The derived classes in this case are the result set of queries, which is similar to the match sets provided by EMF-IncQuery. Note, that while the OCL approach does not offer incrementality, an EMF-IncQuery based approach would.

Derived features. There are several approaches that make extensive use of derived features or provide additional support for their usage.

The PROGRES language [14] allows the rule-based programming of graph rewriting systems. It uses derived attributes for encoding node properties concerning aspects of dynamic semantics. The language includes support for defining how these derived attributes are calculated, and also uses functional attribute dependencies that would allow similar implementation as described in Section 5. However, PROGRES has not been adapted to EMF up to our best knowledge.

The FUJABA [15] tool suite also supports derived edges by path expressions in a non-incremental way.

In [16] Diskin describes a theoretical model synchronization framework that uses derived references for propagating changes between corresponding models. The derived attributes defined in the framework are queries, similarly to our approach, although algebraic and not incrementally updated.

Scheidgen [17] presents a MOF tool that allows the definition of derived features using OCL. It handles derived attributes and operations as custom code provided by the user and redirects calls using reflection, thus incrementality is not supported.

JastEMF [18] is a semantics-integrated metamodeling approach for EMF. It uses derived features as side-effect free operations (i.e. queries) and refers to them as the static semantics of the model. Therefore, our query-based approach could be integrated with JastEMF without problems.

ConceptBase.cc [19] is a database system for metamodeling and method engineering. It allows the definition of active rules that react to events and can update the database or call external routines. Using this functionality, it would be possible to create derived features in models that are updated incrementally based on the data stored in the ConceptBase.cc database. On the other hand, this framework has not been applied in an EMF context.

In a previous tool paper of ours [20], we give an architectural overview of the entire EMF-IncQuery tool where derived features are listed as one of the new features of the tool. The current paper provides all the technical details on using incremental queries for derived features in EMF.

7 Conclusion

We proposed to seamlessly integrate the EMF-IncQuery framework to the EMF infrastructure in order to facilitate the efficient and automated computation of derived attributes and references over EMF models by advanced model queries. Our approach (1) allows to define derived features using an expressive graph-based model query language, (2) offers high performance and scalability thanks to the incremental evaluation technique of EMF-IncQuery [2], and (3) automatically provides notifications to and from derived features which has to be implemented manually in an EMF application.

Future work. Our current research directions include the application of query-based derived features for handling soft interconnections in EMF models and for managing virtual EMF objects derived from query result sets. Furthermore, the EMF-IncQuery framework is under active development, with derived feature support being only one of its many capabilities.

Acknowledgements. We would like to thank E.D. Willink for his suggestions on improving the paper and the anonymous reviewers for their helpful comments.

References

1. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A Graph Query Language for EMF models. In: Proc. of ICMT'11. Volume 6707., Springer (2011) 167–182
2. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: MODELS'10. Volume 6395 of LNCS., Springer (2010) 76–90
3. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Theory and Practice of Model Transformations. Volume 5063/2008 of Lecture Notes in Computer Science., Springer (2008) 107–121
4. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
5. The Eclipse Project: EMF Model Query 2. `http://wiki.eclipse.org/EMF/Query2`.
6. The Eclipse Project: EMFT Search. `http://www.eclipse.org/modeling/emft/?project=search`.
7. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: MoDELS '08, Springer (2008) 53–67
8. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Proceedings of GT-VMT 2009. Volume 18., ECEASST (2009)
9. The Object Management Group: Object Constraint Language, v2.3.1. (Jan 2012) `http://www.omg.org/spec/OCL/2.3.1/`.
10. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST **44** (2011)
11. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. **82**(9) (2009) 1459–1478
12. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: FASE 2009. Volume 6013 of LNCS., Springer (2010) 203–217
13. Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In Stevens, P., Whittle, J., Booch, G., eds.: «UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications. Volume 2863 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 295–309
14. Schürr, A.: Introduction to PROGRESS, an attribute graph grammar based specification language. In Nagl, M., ed.: Graph-Theoretic Concepts in Computer Science. Volume 411 of LNCS. Springer Berlin / Heidelberg (1990) 151–165
15. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. ICSE 2000. (2000) 742–745
16. Diskin, Z.: Model synchronization: Mappings, tiles, and categories. In Fernandes, J., Lämmel, R., Visser, J., Saraiva, J., eds.: Generative and Transformational Techniques in Software Engineering III. Volume 6491 of LNCS. Springer (2011)
17. Scheidgen, M.: On implementing MOF 2.0—new features for modelling language abstractions (2005)
18. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference attribute grammars for metamodel semantics. In Malloy, B., Staab, S., van den Brand, M., eds.: Software Language Engineering. Volume 6563 of LNCS. Springer Berlin / Heidelberg (2011)
19. Jeusfeld, M.A., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. The MIT Press (2009)
20. Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Integrating efficient model queries in state-of-the-art EMF tools. In: Proceedings of the 50th International Conference, TOOLS 2012, Springer (2012) To appear.